A Framework for Automated Exploit Prevention from Known Vulnerabilities in Voice Over IP Services

Abdelkader Lahmadi, Olivier Festor

A Framework for Automated Exploit Prevention from Known Vulnerabilities in Voice Over IP Services

Abdelkader Lahmadi, Olivier Festor

***Abstract*—We propose a prevention system for SIP-based networks which adopts a rule-based approach to build prevention specifications on SIP protocol activities that stop attacks exploiting an existing vulnerability before reaching their targets. Our approach innovates from existing solutions by making use of the contextual information of a vulnerability targeted by an attack to apply the prevention specification. Manually coding these prevention specifications is tedious and error-prone. Our method automatically infers prevention specifications by analyzing captured SIP exploit traffic. The detection engine uses an efficient method based on event graphs to match protocol activities against available prevention specifications. We describe the different components of our approach and show through an extended performance study of the implemented system its applicability to enterprise level VoIP protection.**

***Index Terms*—Exploit Prevention Systems, Security, Session Initiation Protocol, Voice over IP, Vulnerability Management**

I. Introduction

THE Session Initiation Protocol (SIP) [1] is designed to establish, modify, and terminate a session of any application service that uses sessions. The security of the SIP protocol has gained an increasing interest over the years and remains a focal point both in academic and industrial research. Part of this interest is due to the large number of discovered vulnerabilities linked to the SIP ecosystem. The sources of these vulnerabilities are mainly the weakness of some of its implementations, errors in configuration and in rare cases its specification semantics [2], [3]. Both implementation and configuration dependent vulnerabilities are usually discovered using various analysis and testing methods among which, fuzzing [4] where the main purpose is to discover the effect of either malformed messages or the impact of a specific sequence of messages on a particular SIP implementation. Configuration errors can also be identified through an automated security process checking against known vulnerable configurations and best practices. Specification based vulnerabilities can be found through formal analysis of SIP related specifications and can be tested using attack tools.

A. Lahmadi is with the LORIA and Nancy University, Nancy, Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex, e-mail: lahmadi@loria.fr

O. Festor is with the INRIA - Nancy Grand Est Research Center, Nancy, Campus Scientifique - BP 239 - 54506 Vandoeuvre-lès-Nancy Cedex, e-mail: festor@inria.fr

Manuscript received February 28, 2011; revised November 17, 2011. The associate editor coordinating the review of this paper and approving it for publication was James Hong.

A primary way to counter SIP implementation vulnerabilities exploitation is through patching. Patching time however is often long [5] and until the effective update happens the SIP network is kept on leash to attackers. To protect SIP services from such scourge, a first line of defense systems have to be deployed. A defense system can start with network level firewalls, where packets are filtered without a deep understanding of the SIP protocol semantics. Another way, is to use detection engines like *Snort* [6] with a set of known attack signatures.

Recently, several SIP aware application layer firewalls [7], [8], [9], [10] have been developed to counter SIP-based attacks. These solutions to protect SIP networks from known vulnerabilities lack enough flexibility and extensibility. A prevention system needs to be fed with vulnerability specifications together with counter-measures using a domain-specific language. The prevention system runtime screens the SIP traffic and identifies vulnerabilities according to the authored specifications. Most of existing protection systems use low-level languages such as C, or plugins to statically load vulnerability specifications. The authoring process must be repeated for each new discovered vulnerability, despite the fact that many SIP vulnerabilities share common properties over SIP messages, dialogs or fields.

We design a framework with the following goals: (i) provide a language that eases the specification of prevention schemes of discovered vulnerabilities in SIP based services such as Voice-over-IP, (ii) automatically execute the prevention specification by traveling through SIP messages.

In previous work [11], [12], we have presented separately the two core building blocks of our approach: the SecSIP engine and the VeTo language. We present here the overall chain of our automatic prevention framework which includes the former two main components and an automatic facility to generate the prevention specifications. VeTo [11] is an event-driven rule-based language to specify SIP vulnerabilities and their respective counter-measures. It is stateful since it includes features to record SIP protocol states over messages, transactions and dialogs. Each specified prevention scheme is executed by the SecSIP runtime engine deployed as ”a bump in the wire” device in a SIP network. SecSIP uses a graph based approach to process the events and execute the actions specified in VeTo rules. It propagates the events until a vulnerability exploit attempt is detected and a prevention needs to be triggered (e.g. drop a message or reformat an option field). We restrict prevention specification matching to

the vulnerability context within which an attack appears. For instance, an attack may appear within a particular SIP message which targets a specific network element. Development of context aware prevention specifications is another important contribution of this paper. Previous approaches have largely ignored this step when coding prevention schemes.

The remainder of the paper is organized as follows: Section II describes the major known vulnerabilities in the SIP protocol and its implementations. In section III, we present existing SIP specific defense systems and attack specification languages with a focus on the STATL [13] and the SHIELD [14] languages. Section IV details the VeTo language features and its semantics. We illustrate its use on two examples: vulnerabilities coming from malformed messages and exploits using flooding attacks. In section V, we detail our contribution to the automatic generation of VeTo specifications from vulnerable SIP messages. The approach relies on a genetic algorithm applied to regular expressions to characterize malformed messages. Section VI presents the implementation of the SecSIP engine to handle SIP messages and execute VeTo specifications against them. Section VII provides a qualitative analysis of VeTo, Snort and STATL to express different prevention schemes. We then evaluate the SecSIP runtime using basic scenarios to assess its call handling capacity with activated preventions. Finally, we draw some conclusions and identify future works.

II. Background and Motivations

A. Terminology

Some of the terms used in this paper have different meanings in different papers, depending on the author. For clarity and consistency, we adopt the following definitions from the National Information Systems Security Glossary [15]:

- **vulnerability, flaw**: weakness in system security design, implementation, configuration or limitations that could be exploited.
- **attack**: Attempt to gain unauthorized access to a service, resource, or information, or the attempt to compromise integrity, availability, or confidentiality. The success of the attack is not necessary.
- **attacker, intruder, adversary**: The originator of an attack.
- **prevention**: a counter-measure or procedure applied on network activity to stop an attack attempt to exploit a vulnerability, before it reaches its target.

B. SIP basics

SIP [1] is a protocol designed to negotiate, establish and terminate a session between two or more peers. It is used as a signaling protocol to provide many services such as voice, TV or video on demand. The SIP service infrastructure relies on several entities, including a *User Agent* that generates or terminates SIP requests, registrars, where users register themselves and announce their availability in the SIP network and proxies that forward requests in the appropriate SIP networks. Despite the well-known attacks that the SIP architecture inherits through the utilisation of the Internet protocol stack, there are dedicated attacks on the SIP protocol itself or on its implementations. The attacks exploit bugs in SIP implementations or weaknesses in its design.

C. SIP exploits and vulnerabilities

SIP messages are text-based and use the *UTF-8* charset. This feature leads to an easy modification by man in the middle attackers. Cryptographic mechanisms to protect SIP messages are not widely deployed because intermediate nodes that may modify SIP messages for their routing are part of the architectural design of the framework and for tsome of them do not support crypto-based security either by design or through mis-configuration. The SIP protocol uses the *INVITE* message to setup and modify dialogs over their lifetime. According to RFC 3261 [1], the message used to modify dialog properties is called *re-INVITE*, but it has the same method value as INVITE. This may result in a theft of service by exploiting a spoofed message or simply relaying a crafted *re-INVITE* message to establish calls.

SIP routing relies on the proxy elements that take downstream routing decisions based on the routing headers and upstream routing decisions based on *Vias* fields. The SIP routing mechanism leverages several vulnerabilities [16] since any element can insert/delete/alter routing headers and proxies route statelessly without call-route state or global route knowledge. These vulnerabilities may generate different types of attacks. These attacks are broken down into network topology privacy breach, toll fraud and DoS based attacks.

As described in [1], SIP mandates the use of HTTP digest-based authentication as a security mechanism to protect SIP messages. It provides anti-replay protection and one-way authentication to SIP messages. However, the SIP authentication mechanism has several weaknesses [17] since SIP servers and proxies need to access and modify certain fields in crossing messages. This mechanism only applies to a few SIP messages (INVITE, BYE, REGISTER), and leaves other important SIP messages (TRYING, OK, ACK, BUSY) unprotected. It only also protects a few SIP fields (Request-URI, realm), and leaves other important fields unprotected (e.g, From, To).

These weaknesses can be exploited to launch different attacks on SIP based networks. A rich set of existing works [17], [18], [10], [19] have addressed SIP vulnerabilities and exploits to examine how they can be efficiently used to compromise the reliability and trustworthiness of SIP-based networks. While in [17], the focus is made on billing attacks, the SIP protocol is also exposed to traditional DoS attacks [20] on ethernet bandwidth and/or OS/firmware to exhaust available resources. Furthermore, SIP comes with its own set of specific DoS attacks [10] (message flooding, transaction flooding, transaction anomalies). These two contributions, have mainly illustrated SIP exploits rather than the specification of the vulnerability that allows such attacks. In [19], SIP attacks are enumerated together with the vulnerabilities that cause them. The lack of authentication is for example, the major cause of signaling attacks like BYE, CANCEL and Re-INVITE.

D. Complexity of fixing vulnerabilities

In this section, we draw attention to some key aspects of fixing known SIP vulnerabilities, particularly the complexity to be patched and managed adequately.

1) Rising number of vulnerabilities and reluctant administrators: The major problem facing the fixing of vulnerabilities is their rising number and the inability of users to efficiently cope with the number of patches issued for these vulnerabilities [21]. As this number rises, the time to patch becomes high, up to the order of months [5]. In addition, patches can be difficult to apply and might even have unexpected side effects as a result of compatibility issues.

2) Diversity and dynamics of vulnerabilities: Discovered vulnerabilities in the SIP world may involve several dialogs and different headers over multiple messages. Our analysis of existing vulnerabilities shows that they may share the same SIP protocol properties but for different purposes. For example, many malformed SIP message vulnerabilities have the same type of message but rely on different malformed headers like via, call-ID, or content length. A vulnerability has a lifetime dynamics where it may be discovered or disappear after a patching operation or a software upgrade. Therefore, the prevention mechanism requires a configuration change to reflect the evolving nature and the emergence of new vulnerabilities.

3) Multiple vulnerabilities per device: A device involved in a SIP service has its own vulnerabilities due to errors in its implementation. Prevention from exploiting these vulnerabilities has to take into account that one or many vulnerabilities are associated to a particular device. The prevention mechanism has to map vulnerabilities to devices, otherwise it may waste the resources of the prevention runtime. We therefore need to apply prevention specifications according to the existing devices in the SIP network and their respective running state.

III. Intrusion Prevention systems and languages

We present in this section different solutions that have been proposed in the litterature to counter SIP-based attacks and offer prevention languages and supporting runtimes that influenced the design of the SecSIP framework. Traditional solutions like Snort or Bro [6], [22] are general and do not target the SIP protocol. Therefore they are unable to dig into SIP message payloads and interactions leading to limited expressivity. Recently these approaches have started to support some SIP exploits such as INVITE flooding attacks but in a very limited way. Several SIP DoS countermeasure solutions have been proposed to defend a SIP infrastructure [23]. Most of them, however focus only on some specific attacks and their implementations are either partial or unreported.

A. SIP-based defense systems

The existing defense approaches we do consider fall into two categories: proactive and reactive. A reactive defense approach generates an inoculation in response to an attack. This response will protect SIP devices. An example of such inoculation is the application of patches to eliminate a bug exploited by the attacker. There are many additional examples of reactive defense approaches including cross-protocol correlation, statistical analysis, attack signatures, reactive address blacklisting, etc. These solutions attempt to recognise post-attacks and take a counter-measure later. Proactive defense prevents malicious transactions and malformed packets from reaching the intended victim. A common proactive approach to identify malicious behaviour is to record interacting SIP states, attributes and messages, and prevent them from reaching the victim. In this case, the approach is stateful. A proactive approach may operate anywhere in the network perimeter. If located at the victim side such an approach becomes useless, since a denial of service attack damage still occurs while the defense system tries to prevent it. However, if a proactive solution is located at the network ingress point rather than on end-host, the victim is kept unaffected. A proactive approach is suitable to cover compromise attacks like toll fraud, unwanted calls and messages [19].

SCIDIVE is one proposal for stateful intrusion detection [8] propose a solution. The system relies on a stateful detection engine that determines the current state from the observation of multiple packets involved in the same session. The system also uses cross-protocol detection to verify the consistency between two protocols involved in the same VoIP session, mainly SIP and RTP. The mechanism does not provide any mitigation feature; only detection is described. The authors only present the overall approach and a small number of detection rules. While the proposed solution is considered proactive, it lies on UA (victim side) attack detection. Their IDS is host-based, needing to be deployed at a place where it can monitor both SIP and RTP messages. In [7],VoIP defender is presented. VoIP defender is a SIP-based NIDS designed to monitor, detect, analyse and counter attack. It is a building block to be placed as a proactive detection system at the ingress point of the network. Unfortunately, the nature of the employed detection and security scheme (stateful or stateless) is not clearly defined in the available documentation. In addition, no details are provided about the language used to build defense rules and how it can be used for SIP.

In [9], [10], authors propose a SIP attack detection approach based on full state machines specification. They target only attacks on UA (victim side) where an attack signature is specified to describe the state flow in the state machines. The authors have not considered attack mitigation. The solution proposed in [10] is theoretical and needs to be placed into an existing IDS to assess its applicability and performance. In [9], the authors claim an implementation of their IDS without further details and provide only simulation results.

Our designed SecSIP framework shares the stateful feature with SCIDIVE [8] but our approach is more systematic as we formulate how to specify detection and mitigation rules for different attacks. Our solution is designed to be placed as a proactive prevention block at the ingress point of a network which is similar to the VoIP defender solution [7]. The SecSIP framework provides a novel feature that can be coupled to vulnerability and exploit discovery tools. As we described in section V, we developed an off-line tool to generate preventions policies from exploit messages discovered, for example, by the KIF [4] fuzzing tool. These generated policies can then

be deployed on the SecSIP prevention engine.

B. Preventions enabling languages

To the best of our knowledge, no domain specific language has been published so far to model SIP attacks or vulnerabilities and their respective counter-measures. However, multiple research activities [23] were focused on the development of runtime systems and mechanisms where vulnerability occurrences and attack signatures are encoded internally. In the general area of network intrusion detection mechanisms, several attacks and detection languages have been proposed [24].

We identify two classes of such languages: stateful and stateless. Stateless languages describe attack events independently, while stateful languages consider the relationships between events and are able to model event histories that represent attacks. Existing languages do not provide features to address the SIP protocol semantics necessary to describe vulnerabilities. Instead, they use generic signature models, e.g., applying regular expression or state machines to a known protocol data. The generated signatures are usually generic and do not consider the context properties of a vulnerability.

1) Stateful languages: Languages based on finite state machines such as STATL [13] and SHIELD [14] have been designed to express network protocol attacks and vulnerabilities. SHIELD is a vulnerability driven language used to express application layer binary protocol vulnerabilities at end-hosts. In their approach, the authors only specify the part of the protocol state machine that exhibits a vulnerable behaviour. VeTo shares with SHIELD this feature since our aim is to only specify the part of the SIP protocol and its instances involved in a vulnerability. Due to the limited publicly data available on the SHIELD specification, we could not assess its practical applicability to SIP.

GAPAL [25] is a generic application level protocol analyser that relies on a stateful language to describe protocol syntax and semantics for their parsing and analysis. GAPAL and SHIELD have the same syntax, the former being an extension to the later to support the analysis of text-based protocols.

In [13], the authors present a state/transition based language called STATL, dedicated to attack recognition. The language allows the description of domain-independent attacks. It relies on the state transition analysis technique [26] which is an abstraction of attack actions to a higher representation. In STATL, an attack is modeled as a sequence of steps that bring the system from an initial safe state to a compromised state. The attack is represented as a composition of *states* and *transitions*. They also use variables to record the part of the system state needed to define an attack signature. Each transition has an associated event that may cause the transition to a new state. The language has been used on different domains such as network attacks and Unix system attacks. Unlike STATL, the VeTo language is not a pure state/transition based language, but it uses an event-driven approach. An event-driven approach is more suitable to overcome the STATL limitations regarding time, intervals and relationships among events. But representing events as transitions through a single chain of states precludes the recognition of a vulnerable behaviour without any time order. In addition, a pure state/-transition approach is ineffective [27] and may introduce a state explosion problem when describing a large number of vulnerabilities.

2) Stateless languages: Snort [6], the defacto-standard open source intrusion detection tool relies on a signature-based language to detect intrusion. Its language is stateless since it does not provide means to record protocol states. The stateful feature of Snort is provided by the preprocessors, where the inspection is hard coded. For example, the *stream4* preprocessor is used by Snort to maintain an internal state table for each TCP session. The VeTo language provides an expressive way to construct stateful variables which keep track of SIP protocol histories. In addition, the language provided by Snort is signature-based and relies on exploit description rather than the vulnerability description approach used by the VeTo language.

Extending these languages to support the SIP semantics does not solve these issues since their underlying matching models are hardcoded. Therefore, we designed a new language dubbed VeTo dedicated to the prevention from existing vulnerabilities exploitation in the SIP protocol.

IV. VeTo language overview

VeTo relies on three features to counter a particular known vulnerability exploit. The language combines a context, a definition and the event properties of a vulnerability to provide the ability to prevent against its exploitation. The context block exhibits the vulnerability surrounding environment properties. The definition block provides the vulnerability related assumptions on its behaviour such as the involved SIP messages and their respective fields. The prevention block describes the vulnerable behaviour within its context and includes a response action. The definition and the context blocks can be shared by different vulnerabilities. However, each vulnerability has its own prevention block.

A. Context block

The context block describes the information associated to different specified protection blocks. In VeTo, a context is defined as a set of labeled attributes with predetermined values. Instead of a value, an attribute can have a set of values. The context attributes are mainly the URI of the targeted SIP entity, the date at which a patch is expected to be applied to remove the vulnerability and the firmware version where it is reported. The context is used to trigger the proper protection block according to the current information available to the language runtime. A context is associated to one or several protection blocks, but a vulnerability has a single context block.

Rule 1 shows a specification of a context block.

The `TARGET` attribute denotes a SIP element that is concerned by a vulnerability protection scheme. A target value is composed of the transport protocol underlying the SIP traffic behind the vulnerability, an IP address, range or hostname of the targeted SIP element and the port of that SIP traffic. The `LIFETIME` attribute is specified as recommended in RFC

Rule 1 A VeTo context specification.

```
CONTEXT GlobalCtx BEGIN
TARGET => udp:phone1.example.com:5060;
TARGET => tcp:phone2.example.com:*;
TARGET => *:softphone1.example.com:*;
CONTEXT END
CONTEXT myCtx BEGIN
INCLUDE => GlobalContext;
LIFETIME => 2009-05-07;
CONTEXT END
```

3339. It denotes the date when the vulnerability is announced to be fixed or removed. When this date is unknown, the attribute is omitted from the context block. The `INCLUDE` property allows to include the content of another context. For example, consider the context block called `GlobalCtx` that enumerates a list of sip devices using the target attribute. Then, there is a context called `myCtx` that includes `GlobalCtx` and contains a lifetime attribute to specify the patch time of a particular vulnerability.

B. Definition block

The definition block relies on regular expression based pattern matching rules. Each rule is composed of an optional header part including a pattern matching statement and a mandatory action part. The header of each rule is the composition of terms followed by the *@MATCH* operator and a regular expression. The body of the rule is an action which defines typed variables to be used by a protection block. The term component refers to an element from the parsed tree of a SIP message. When the term value matches the given pattern, the variable is created by the runtime.

Rule 2 shows a definition block named *contactDefs*. The block contains two rules. The first rule defines a *SET* type variable used to store the values of the *contact* field of a SIP message. The contact field is referenced using the predefined constant *sip:headers.contact*. The second rule contains a matching pattern against the *method* field of a SIP request. If the field matches the pattern INVITE, the action creates an event named *ev_Invite*.

Rule 2 A VeTo definition block.

```
DEFINITION contactDefs BEGIN
LET: SET[sip:headers.contact] contacts;
WHEN sip:headers.method @MATCH "^INVITE$" ->
                        LET: EVENT ev_Invite;
DEFINITION END
```

C. Prevention block

The prevention block relies on event-based rules which describe the logical part of the vulnerable behaviour. It specifies the event patterns and their respective actions when they are satisfied. The event pattern part uses the set of variables defined in the definition blocks. Each prevention block specifies a protection scheme and has a unique identifier, an optional context specified @ symbol and a `USES` statement to use a set of definitions. Rule 3 shows a protection block named `myProtection` with a single rule. The block has as context `myCtx` which was described earlier. The rule header part specifies an event pattern based on the event `ev_Invite`. The occurrence of the event triggers the `STORE` action against the previously defined set type variable `contacts`. The `STORE` action keeps in memory the collection contacts which contains the values of the contact field over observed SIP messages.

Rule 3 A VeTo prevention block specification.

```
VETO myProtection@{myCtx} USES ContactDefs BEGIN
(ev_Invite) => STORE:contacts;
VETO END
```

1) Events Sequence: The events sequence takes as input a list of n event labels. It specifies a particular order in which the events must occur to be considered.

The following example represents a simple pattern of two events labels *ev_ack* and *ev_invite*. The rule checks if the *ev_ack* event precedes the *ev_invite* event. If the pattern is matched, then the SIP message is dropped.

```
(ev_ack,ev_invite) -> DROP;
```

We note that we can use short cut notations to describe event sequences. We can use, for example the repetition operator "*" as described below.

```
(ev_invite[*2],ev_200_OK,ev_ack)
```

The above pattern checks if two successive *ev_invite* events are followed by *ev_200_OK* and *ev_ack* events.

2) Embedded events: An embedded event is an event that occurs at the same time as another event. It allows to match an arbitrary number of events that appear at the same instant of the arrival of a SIP message.

3) Negation patterns: A negation pattern specifies an event that does not appear within a sequence of events. When the sequence is empty, the negation pattern specifies any event that is different from the specified event. The negation pattern is denoted by the symbol $\sim$. Rule 4 depicts the usage of a negation pattern. The definition block creates the variable `contacts` of a set type as depicted in line 2. It also creates an event named `Ev_BYE` as depicted in line 3. The rule in the Veto block checks the non occurrence of the event `Ev_BYE` to feed the contacts list with the values of the field `contact` of a SIP message.

Rule 4 An illustration example of the usage of a negation event pattern.

```
1
2  DEFINITION NegationDefs BEGIN
3  LET: SET[sip:headers.contact] contacts;
4  WHEN sip:request.method @MATCH "^BYE$" ->
5                          LET: EVENT Ev_BYE;
6  DEFINITION END
7
8  VETO Negation USES NegationDefs BEGIN
9  (~Ev_BYE ) -> STORE: contacts;
   VETO END
```

4) Temporal patterns: VeTo provides a time window operator to express a temporal relation between events within a sequence. Its aim is to represent the fact that some statements are only true over a given period of time. The digit operand is the time window value over which the pattern has to be matched. After this time window, the pattern is invalid. For example, in Rule 5 the VeTo block stops a flooding attack as soon as 1000 INVITE messages arrived in less than one second time window.

Rule 5 The VeTo protection block against a flooding attack using temporal event patterns.

```
DEFINITION SIPMessages BEGIN
WHEN sip:request.method @MATCH "^INVITE" ->
                        LET: GLOBAL EVENT ev_invite;
DEFINITION END

VETO Flooding USES SIPMessages BEGIN
([ev_invite[*1000],1]) -> DROP;
VETO END
```

D. Variables scope and extent

In *VeTo*, each variable has a scope and an extent. The scope determines where the variable and its value are associated. The extent determines when the value is associated to the variable at runtime. The scope of a VeTo variable is related to its definition block. A VeTo variable is visible over all protection blocks that reference its definition block.

The extent of a VeTo variable is either message or dialog. A SIP dialog is defined using the tuple (*Call ID, To Tag, From Tag*) fields of a SIP message. Typically, the predefined constants named as *sip:request.**, *sip:response.** and *sip:headers.** have an extent over a message. They take a new value each time a new SIP message is observed and parsed. The variables defined into a definition block and used by a prevention block have an extent over a dialog. A variable that takes its values over multiple dialogs is defined using the keyword `GLOBAL` followed by its type and name.

E. Illustrative Examples

We illustrate the use of the VeTo language over well identified SIP vulnerabilities found in the literature. We have selected three types of vulnerabilities with different degrees of complexity. The vulnerability preventions described below share the definition block depicted in Rule 6.

1) Malformed messages: VeTo provides the capabilities to address malformed messages according to two approaches [28]. The first approach is the misuse prevention, where the rules describe a known discovered malformed pattern within a given SIP message. For example, it is reported on several VoIP security mailing lists [29] that many hardware SIP phones are sensitive to SIP messages with illegal size fields value. The protection block described in Rule 7 depicts a prevention against a vulnerable SIP message where the size of the date header is greater than an allowed size of 120 bytes. The size of the date field is computed using the symbol *&* that precedes the field name *sip:headers.date*.

Rule 7 A prevention against a vulnerable *OPTIONS* message.

```
VETO IllegalSize USES SIPDefs BEGIN
(ev_Options) -> IF (&sip:headers.date @GE "120")
{
          DROP;
}
VETO END
```

The second approach is a specification based description of SIP messages. In this approach a set of VeTo rules enforce the ABNF of SIP messages as they have been specified in RFC 3261. The advantage of this approach is that it protects a SIP network from unknown or undiscovered malformed patterns that may crash deployed SIP implementations. However, using this approach, numerous false negatives may occur since some implementations introduce their own specific SIP fields that do not exactly follow the SIP specification. Rule 8 depicts an example of an INVITE SIP message with a non compliant *Call-ID* header that will be dropped. The rule in the definition block *MalformedDefs* creates the event $ev\_malformed$ when a Call-ID header does not match a compliant Call-ID regular expression.

Rule 8 A prevention against a non compliant Call-ID SIP messages field.

```
VETO EnforceSpec USES SIPDefs,MalformedDefs BEGIN
(ev_Invite(ev_Malformed)) -> DROP;
VETO END
```

2) Flooding attacks: In [10], the author was interested in SIP-specific DoS attacks by flooding SIP entities with SIP compliant messages. He proposes a detection method that relies on thresholds. The trivial one is a threshold on the number of INVITE messages allowed per dialog to a particular URI. Rule 9 describes the prevention block specifying this type of threshold. Firstly, we track the target URI of each INVITE message using the *STORE* action that acts on the set type variable *targets*. We count the number of *INVITE* messages sent to each target from the collection using the action *APPLY* that acts on the counter type variable *targets.count*. The *targets.count* will be decremented by 1 every 2000 milliseconds. Finally, we check the number of observed INVITE messages sent to a specific target over all the existing dialogs against an allowed threshold. If the threshold value is crossed, we drop all subsequent INVITE messages.

The most interesting threshold is the upper bound of the number of allowed transactions per node. Firstly, we define a collection that tracks the transactions per node. We also define a counter to track their number. Rule 10 describes the prevention block for this vulnerability.

3) DoS using broken handshaking: This attack is based on broken SIP handshaking where the attacker sends an INVITE request and then ignores the 200 OK response refusing to send the ACK. The attacker proceeds with a large number of broken initiations in order to exhaust the target resources. Rule 11 shows the prevention block for this vulnerability. The first event rule in the veto block `VulHandShacking` defines

Rule 6 Examples of VeTo definition blocks.

```
DEFINITION SIPDefs BEGIN
WHEN sip:request.method @MATCH "^INVITE$" -> LET: GLOBAL EVENT ev_Invite;
WHEN sip:message.method @MATCH "^OPTIONS$" -> LET: EVENT ev_Options;
WHEN sip:request.method @MATCH "^ACK$" ->  LET: EVENT ev_Ack;
WHEN sip:request.method !@MATCH "^$" -> LET: EVENT ev_Request;
WHEN sip:response.code @MATCH "^200" -> LET: EVENT ev_200OK;
DEFINITION END

DEFINITION MalformedDefs  BEGIN
WHEN sip:headers.Call_ID !@MATCH "^\s*(Call-ID|i)\s*:\s*w+[@w+]$" -> LET: EVENT ev_Malformed;
DEFINITION END
```

Rule 9 Prevention from an INVITE flooding attack against a particular target URI.

```
DEFINITION FlDefs BEGIN
LET: GLOBAL SET[sip:headers.uri.addr] targets;
LET: GLOBAL COUNTER(1,2000) targets.count;
DEFINITION END

VETO FloodingTarget USES SIPDefs,FlDefs BEGIN
(ev_Invite) -> {
        STORE:targets;
        APPLY:targets.count;
        IF (targets.count @GE 10) {
            DROP;
        }

}
VETO END
```

Rule 10 Prevention from a transaction based flooding attack.

```
DEFINITION FlDefs BEGIN
LET: SET[sip:headers.branch] transactions;
LET: COUNTER(1,60000) transactions.count;
DEFINITION END

VETO FloodingTr USES FlDefs,SIPDefs BEGIN
(ev_Request) -> {
         STORE: transactions;
         APPLY: transactions.count;
         IF (transactions.count @GE "10") {
            DROP;
         }
}
VETO END
```

an event pattern to detect a 200 OK response without any ACK message arriving within a time window of one second. In such case, it feeds the set type variable `blackList` with the URI of the sources of the uncompleted handshake messages. This list is used by the second rule from the same VeTo block to prevent the attacker from reaching targets by sending more broken handshakes or SIP messages. The symbol * is a Kleen closure to denote the occurrence of any event within the current dialog.

V. Automatic Generation of Prevention Specifications

Building and maintaining prevention rules using the VeTo language may be a time consuming and error prone task, especially when addressing an important number of vulnerabilities discovered using a fuzzing tool. The discovered vulnerabilities using such process are usually based on a single exploit

Rule 11 Protection block against the broken handshaking SIP attack.

```
DEFINITION HandSDefs BEGIN
LET: SET[sip:headers.from] blackList;
DEFINITION END

VETO VulHandShacking USES HandSDefs,SIPDefs BEGIN
(ev_Invite,*,(ev_200OK,~[ev_Ack,1])) -> {
     STORE: blackList;
}
(*) -> IF (blackList @contains sip:headers.from) {
         DROP;
}
VETO END
```

message with a malformed field or sequence of vulnerable messages. To reduce this effort, we have designed a generation method to produce VeTo specifications targeting those vulnerable messages. The method mainly characterizes a malformed field within an exploit message or the vulnerable sequence of messages and generates a set of VeTo rules specifications to prevent their exploit. The generated VeTo rules are then deployed and maintained on the SecSIP engine to be applied against the SIP traffic exchanged between the outside network and the inside network where SIP devices are deployed. Figure 1 depicts the overall architecture of our solution including the automatic generation of VeTo specifications.

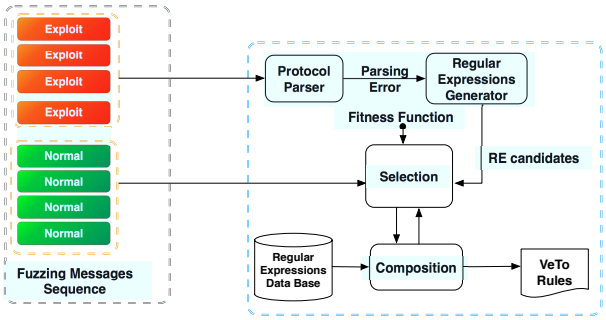


Fig. 1. Overview of modules and processing flow for automatic generation of VeTo specifications.

The solution relies on generating a set of candidate regular expressions to match a malformed pattern within a SIP message, and evaluate their quality to ensure that their are specific enough to only match exploit messages. As input, the generation algorithm takes only a set of labeled SIP messages which are tagged as *Exploit* or *Normal*. Exploit messages are

typically obtained from a fuzzing tool and assessed to incur an attack on a specific SIP device. In this work, we have used the KIF fuzzing tool [4] to obtain such sequences of labeled messages. For each exploit message, we invoke a SIP parser which is generated by ANTLR (a parser generator) from the SIP protocol specifications (ABNF rules of RFC 3261). Whenever an exploit message is parsed, a parsing error is raised and indicates the malformed field identifier within the exploit SIP message. The parsing error is due to a violation of the SIP protocol syntax rules or an unknown field. Parsed exploit messages are grouped according to their respective malformed fields. Each group contains messages that share the same malformed field identifier but with different malformed strings.

After parsing, we generate from each exploit message within the same group, a set of regular expressions (regex) that match the malformed field. This set of regular expressions is generated according to Algorithm 1 which takes as input the malformed string identified by the SIP parser.

Algorithm 1 Candidate regular expressions generation.

```
if it's a number then
    replace with \d or keep the token (0.5 prob)
else if it's a character then
    replace with \w or keep the token (0.5 prob)
else if it's a sequence of numbers then
    replace with \d+
else if it's a word then
    replace with \w+
else if it's a reserved SIP protocol tag then
    replace with ("sip:", "From" , "To", etc)
else
    keep the token as a literal regex;
end if
```

The generated set of regular expressions which match a specific malformed field are denoted as candidate regular expressions. Figure 2 shows an example of a malformed SIP message and the set of candidate regular expressions. Herein, the malformed field is the request URI which contains a non compliant character `\255`.

In a second step our algorithm identifies the group of regular expressions which best characterize the underlying malformed field with respect to normal messages.

A. Regular expressions optimization

Given the set of candidate regular expressions, we derive the best regular expressions which only cover exploit messages. Regular expressions matching a normal message are removed from the candidate set. We use a genetic algorithm [30] to optimize the set of candidate regular expressions. The algorithm considers each candidate *regex* as a chromosome which is evaluated using an appropriate fitness function. A selection algorithm is then applied to select the best *regex*. Finally, we apply a set of recombination and mutation rules on the selected *regex* to generate a new generation of regular expressions. The cycle of evolution is repeated until a defined

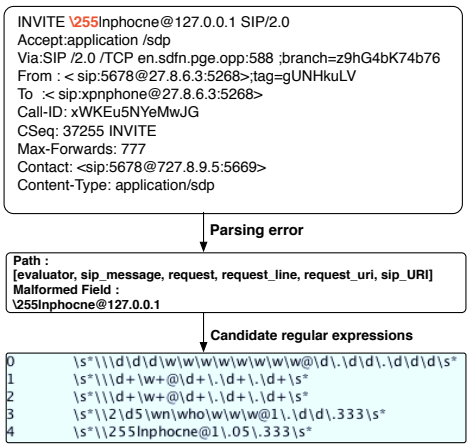


Fig. 2. Example of a malformed SIP message and the set of candidate regular expressions.

number of evolutions is reached. In our work, we fixed this number to 10 generations. The overall process as specified by Algorithm 2 is repeated until we get the best set of regular expressions.

Algorithm 2 Genetic Regular Expression Generation.

```
Generate a set of Regex using predefined rules
for i from 1 to number of generations do
    Test the set of regex against normal and exploit messages
    Assign a score to each regex using a fitness function
    Apply the selection algorithm
    Recombine regex of selected parents
    Apply mutations on the new population of regex
end for
```

We have defined a suitable fitness function to evaluate the quality of each generated regex. Our fitness function combines the number of exploit messages covered by the regex and its length. The idea behind, is that the best regex covers many exploit messages and has a short length. The fitness function f is defined as follows:

$$f(regex) = n \times (1 + (\frac{avg - l}{avg})) \quad (1)$$

where avg is the average length of the regular expressions, n is the number of exploit messages covered by regex and l is its length.

Our genetic approach to generate regular expressions relies on a selection algorithm to determine the best candidate to be parent of new generated regular expressions. Algorithm 3 details the selection process.

The last mechanisms used by our genetic algorithm to generate optimized regular expressions are the reproduction operations. These operations are used to recombine selected parents to generate new regular expression. We have used two operations depicted in Algorithm 4.

The first operation combines the parents regular expressions without mixing them using the OR, AND and + symbols. The second operation mixes the two parents regular expressions

Algorithm 3 Selection algorithm.

P: the vulnerable pattern identifier
MMS(P) = set of exploit messages with respect to P
CRES(P) : set of candidate regular expressions with respect to P
CRES(P) =
while MMS(P) not empty **do**
  Evaluate the score of each regex
  Select the regex with the higher score
  Add the selected regex to CRES(P)
  Remove from MMS(P) all exploit messages covered by the selected regex
end while

Algorithm 4 Examples of reproduction operations.

First operation: X|Y, XY, X+Y?
Second operation: crossover
X: SIP:XXX@\d.\d.\d.\d. ppp
Y: SIP:YYY@\d.\d.\d.\d. fff
XY: SIP:(XXX|YYY)@\d.\d.\d.\d (ppp|fff)

by using a crossover operator to obtain a mixture of them. While the genetic process terminates and we obtain the best set of regular expressions which only matches exploit messages, the algorithm generates a set of VeTo rules to specify events over these regular expressions. These events are then used to generate prevention blocks as described in section IV.

VI. SecSIP Implementation

In this section, we discuss various implementation details of SecSIP as a runtime of authored VeTo specifications to be applied against SIP traffic. The overall architecture of SecSIP [12] is depicted in Figure 3.

The role of the **Input/Output Layer** is to capture packets from the network, and to pass them to the SIP Parser. Once the processing of the packet by the SecSip Rules Engine is finished, the Input/Output Layer is instructed to either release the packet into the network, or to discard it. The **SIP Parser** analyses the packet, and extracts the value of relevant SIP fields, it then hands over the extracted data to the EventController, in the form of a SIP Tree. The **Event Controller** uses event definitions from the loaded Rules Specification, and generates events in accordance with the presence, or value of certain elements of the SIP Tree. The **VeTo Rules Specifications** are data structures that represent sets of filtering rules, in an un-compiled format, which allows them to be easily exported and imported to SecSip. When a Rules Specification is loaded by the Rules Engine, it is first compiled, in order to offer optimized execution. The **Rules Engine** is the main component of SecSip. Its role is to track the state of the SIP dialogs that go through the firewall, and to block them as soon as an attack pattern is detected. To do so, it uses a graph compiled from a VeTo Rules Specification, and events generated by the SIP Parser. The state of each active dialog is stored internally by the engine. The **HTTP Server and CLI Agent** provide a way for administrators to interact with SecSIP while it is running, either monitoring its status, or changing the filtering rules. The CLI Agent provides a shell-based interface to telnet clients, whereas the HTTP Server provides a web-based interface to web browsers, which can communicate with SecSIP through SOAP requests. The **SOAP Server** handles requests from SOAP clients, converting VeTo specifications from and to XML trees.

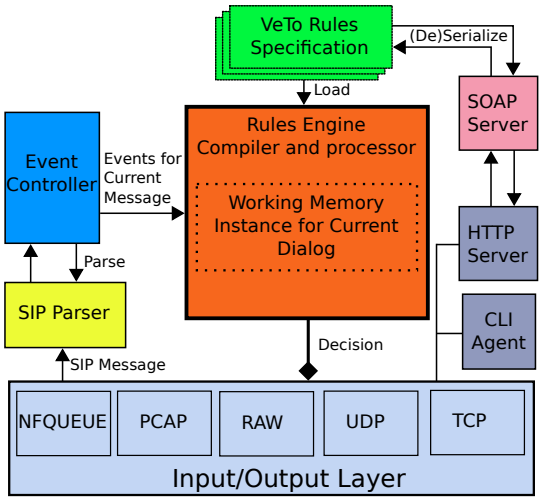


Fig. 3. Functional architecture of the SecSIP engine.

A. SIP messages handling

When a packet reaches SecSIP, it first has to be intercepted so that it can be processed. The packet is then passed to a SIP parser, which grammatically decomposes it in SIP and SDP fields. These fields can then be matched against regular expressions defined in VeTo rules, and events are then generated. The generated events are fed into the event graph as tokens. The tokens are propagated through the graph until an action is reached, or until the graph state does not allow them to be propagated any further. Actions that have been reached are executed, and they can cause the packet to be dropped. Finally, the packets that were not dropped are released, and routed into the network.

The first step of SIP packets filtering is their evaluation against definition blocks. These blocks allow to declare the variables that are used to track the state of a dialog and the different known vulnerability schemes. By default, these variables are tied to a specific dialog, but their scope can span all dialogs observed by SecSIP, when declared with the *GLOBAL* keyword. Variables can be of several types and can be static or dynamic. Static variables take their value from SIP and SDP fields or are explicitly defined in context blocks, while the value of dynamic variables is set by logic (inherent to the type of the variable, or set by actions contained in VeTo blocks).

A variable has one of the following types:

- **collection**: a collection variable is declared with the name of a SIP field as a parameter. Each time the variable is updated, the value of the field for the current packet is

added to the collection. It is then possible to check if a specific value is contained in the collection,
- **counter**: a counter variable starts with a value of 0, and is increased each time the variable is updated. A counter variable also has a "cooldown" facility: the counter can be periodically decreased by a user-defined amount,
- **timer**: a timer variable has a user defined value, and when set, will start to count down until it reaches zero,
- **state**: a state variable can be used to implement a user defined state machine, and to take actions based on the current state,
- **event**: an event variable is a flag that can be raised conditionally. It is used by SecSIP to match patterns defined in VeTo prevention rules.

B. Event patterns handling

In order to track the event patterns, we use an event graph, inspired by active databases [31]. In this section, we detail the structure of the graph, then explain how it is built, and finally go through the execution of the graph.

1) Graph structure: Building one single graph for the complete set of rules allows SecSIP to factorize as much as possible of the event patterns, and reduce the number of necessary tests. All events or sub-patterns that are common to several event patterns only have to be tracked once.

This way, instead of having to maintain one state machine per event pattern per SIP dialog, we only maintain one global graph for all dialogs, which also reduces memory usage.

The event graph is represented using three main structures: graph nodes, links between nodes, and tokens (which represent the flow of events through graphs).

In the event graph, each node represents an event, or a composition of events. Single events are the bottom nodes of the graph with only outgoing edges. Each node above and up to the top adds a level of composition. At each top node of the graph with only incoming edges, we have one of the patterns that the graph represents, and the list of conditions and actions associated with the pattern.

Nodes are connected by links, which also carry a meaning. Therefore two nodes can be connected by multiple links with different meaning, depending on the destination node type.

Each node also carries a list of tokens, which represent events or compositions of events as they travel through the graph. Tokens originate from the bottom nodes of the graph each time an event is "raised"; and each time a token reaches one of the top nodes, the conditions and actions referenced in this node are processed.

Rules 12 show an example of set of rules whose event graph is described in Figure 4.

- bottom nodes of the graph represent simple events. "*" is the "any" event, which always generates a token, for each received packet, and "`~ev2`" is a negative event, which generates a token when event "`ev2`" was not triggered by the current packet,
- action nodes are on the top of the graph. Each action node contains a reference to one of the Veto rules (in this example the three rules are only composed of the "drop" action),
- links between nodes are of different colors, depending on the type of the link: left (L), right (R), start (S) and trigger (T),
- the "^" node represents the embedded event "`(ev1(~ev2))`" which means that "`ev1`" and "`~ev2`" must happen at the same time.
- the "`{5}`" node is a repeat node, meaning that the left token `(ev2)` must be repeated exactly 5 times to be propagated,
- the "`(,3)`" node is a timer node, meaning that its left token `(ev2)` must happen within three seconds after the preceeding token `(ev1)`. The "any" event is used as a trigger in the cases we negate the timer, and we don't want the left token to be received within the specified time,
- the "," node is a composite node, meaning that its left token `(ev1)` must happen right before its right token `((ev2, 3))`.

Rule 12 An example set of VeTo prevention rules.

```
(ev1, [ev2, 3]) -> DROP;
(ev1(~ev2)) -> DROP;
(ev2{5}) -> DROP;
```

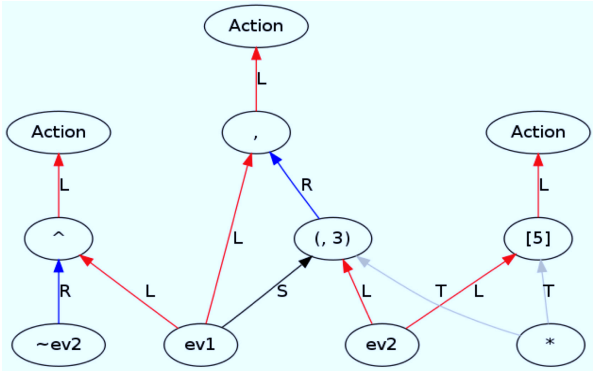


Fig. 4. The event graph produced from the set of rules defined in Figure 12.

Each token carries three pieces of information:

- the **start packet number**, which is the sequence number of the earliest packet for which the token is valid (inclusive),
- the **end packet number**, which is the sequence number of the latest packet for which the token is valid, plus one (exclusive),
- a **reference counter**, which is increased each time a token is pushed through a link, and decreased each time the token is discarded. When it reaches zero after discarding a token, the memory used by the token is freed.

The sequence number of a packet is determined by a per-dialog counter. Each time a packet is captured, the value of the counter is associated with the packet, and the counter is incremented. The idea behind this is that tokens should represent intervals for which events (or sequences of events) are valid. So, when a token hits an action node, its interval

will span the complete attack pattern that was described in the configuration. For instance, if event ev1 was raised by packet 0, and event ev2 was raised by packet 1, two tokens would have been created: one with interval `[0, 1[` for ev1 and one with interval `[1, 2[` for ev2. If those two tokens were to be pushed to a composite node expecting ev1 to happen right before ev2, it would discard those two tokens, and create a new one, with interval `[0, 2[` and push it further through its links to other nodes of the graph.

Each node may contain a list of links. One link contains a reference to the node it connects to, as well as the type of the link. Link types are as follows:

- **left** and **right link** are "normal" links, through which tokens usually travel. Their meaning is not necessarily different. For instance, links to an embedded node (in which tokens have to be simultaneously propagated), which token came from left or right does not matter, whereas in composite links, the left token has to happen before the right token,
- the **tick link** is used by nodes that are not only triggered by the tokens they control. In the case of a timer node, the node might have to time out if no token was pushed to it, in which case a left or right link is not an option,
- the **start link** is used to inform the timer node when the timer has to start, and this cannot be done by a left or right link, as the timer has to start before the token it controls is first received.

Nodes of the graph represent single events (bottom nodes), or composition of events that represent the event patterns. Each top node of the graph represents one of these complete patterns that trigger rule actions. Each time a token is pushed to a node, it is queued for execution. Each type of node has its own execute function, that determines how and when tokens should be propagated. The following types of nodes are used:

a) Bottom Node: A bottom node produces a token each time the referenced event is detected in a packet (or absent from it, if the negative flag of the node is set to "true").

b) Action Node: An action node executes the rules that it references, each time a token is passed to it. Action nodes are the leaves of the graph. A rule consists of a condition (optional) and of one or more actions. If there is no condition, or if the condition is true, the actions are executed sequentially. An action node takes its input from a left link.

c) Composite Node: A composite node takes input from one left link and one right link. If a token from the left link immediately precedes a token from the right link, they are merged in a new token (union of the two previous tokens), which is then propagated to children of the current nodes.

d) Embedded Node: An embedded node takes input from one left link and one right link. If a token from the left link overlaps with a token from the right link (meaning that the events happen at the same time, at least partially), they are merged in a new token (intersection of the two previous tokens), which is then propagated to children of the current nodes.

e) Repeat Node: A repeat node has a minimum, and an optional maximum value. It takes its input from a left link, and a tick node is used to trigger the execute function each time a packet is received. When the count of packets from left link is within the min and max range, a new token is propagated, which spans the matched repeat sequence.

f) Timer Node: A timer node controls wether a token arrives within a certain period of time, defined in seconds. When it receives a token on its start link, it registers the time at which it arrived, and it will forward the first token it receives on its left link only if it is received within the defined period. It can also be set to negative, in which case it will create a new token and propagate it if no token was received on the left link during the defined period of time. In this case, it uses a tick link to be triggered each time a packet is received.

2) Building the Graph: Rules are added sequentially to the graph. Each time a node insertion is required, SecSIP first checks if a similar node already exists in the graph. If such a node exists (same parameters, same parents), it is simply reused.

Once each node has been added to the graph, the path length of each of them is computed. It is set to zero for action nodes, and increased by one for each traversed node. For a node that has multiple paths to action nodes, the highest value is used as the path length.

Then, a list of bottom nodes, sorted by decreasing path length is built.

3) Event Graph Execution: When a packet is received, SecSIP first updates the events defined in the definition blocks, as seen previously. Then, the *execut* function of each bottom node is called.

If the execution of a node causes a token to be produced, it is propagated to other nodes, using the list of links held by the executed node. The propagation is done by inserting the destination node into a triggered node list, sorted by decreasing path length.

Then, each node in the triggered node list is executed, which can cause new nodes to be added to the list, until the list is empty.

4) Actions Execution: Action nodes are top, or terminal nodes of the graph, they do not produce tokens, but instead hold a list of rules, each of which can contain a condition, and one or more actions.

When an action node is executed, it processes its rules sequentially. First, it checks if a condition is present and if it is true, then it executes each action defined in the rule.

The following action types exist:

- the **STORE** action is used to add a value to a collection, using the value of a SIP field of the current packet,
- the **ASSIGN** action is used to set the value of a singleton, state, counter, or timer variable, using either the value of a SIP field of the current packet, or a used defined value,
- the **APPLY** action is used to update the value of a counter or timer variable, using the logic inherent to the counter or timer types,
- the **DROP** action is used to discard a packet, to prevent it from reaching its target.

VII. Evaluation

In this section, we present the evaluation of the VeTo language regarding its expressive power and the performance

of its underlying prevention tool.

A. Qualitative analysis

We have compared the complexity of the VeTo, STATL and Snort languages regarding the number of line of code required to specify the different vulnerability exploitation preventions described in section IV-E. The result is given in Table I.

Vulnerability	STATL	Snort	VeTo
IllegalSize	8	1	3
EnforceSpec	7	1	2
FloodingTarget	22	1	8
FloodingTr	22	-	8
VulHandShacking	21	-	7

TABLE I
SIZES (LOC) OF STATL, SNORT AND VETO SPECIFICATIONS FOR PREVENTING AGAINST DIFFERENT VULNERABILITIES.

We note that we only counted the rules and the event block headers for VeTo specifications. We observe that the Snort language is not suitable to prevent from the exploitation of all vulnerabilities due to its stateless nature. However, it is useful to prevent simple attacks like buffer overflow or flooding. Snort based preventions are limited since a signature specifies the attack occurrence rather than the vulnerable behaviour of the SIP protocol. The STATL language specifications are up to 3 times longer than the corresponding VeTo specifications, because each vulnerability needs its own state machine to be completely described.

B. Experiments

We have setup a testbed of 3 machines each with a core 2 CPU cadenced at 3.4GHZ and 2 GB of RAM. The three hosts are connected through a 100 Mbits switched Ethernet. One host acts as a SIP proxy running version 1.4 of an OpenSIPS server. The two others act as SIP user agents (UA). The SIP UAs are implemented using the *SIPp* [32] tool which allows to write customised SIP protocol interactions as XML documents. The SecSIP tool [12] acts as a support runtime of the VeTo language and implements required features to intercept SIP message in real time and executes prevention specifications. Our performance metrics are the number of established calls and their respective establishment delays. A call is established when the 5 SIP messages *INVITE*, *100*, *180*, *200* and *ACK* are delivered correctly between the two UAs. We compared the performance of SecSIP with the Suricata IPS tool [33] which is able to run Snort based rules.

1) Base capacity and overhead: We have first measured the performance of the SIP proxy and the UAs without any prevention support neither SecSIP nor Suricata. In a first experiment, we have deployed SecSIP and in a second one we have deployed Suricata on the same host as the SIP proxy to intercept SIP traffic using the *nfqueue* interface and running the prevention rules. In each experiment, we varied the number of call attempts from 10 to 1000 INVITE/second with a step of 100. SecSIP runs the prevention block *EnforceSpec* depicted in Rule 8. Suricata runs a single rule which applies a simple regular expression on an INVITE message.

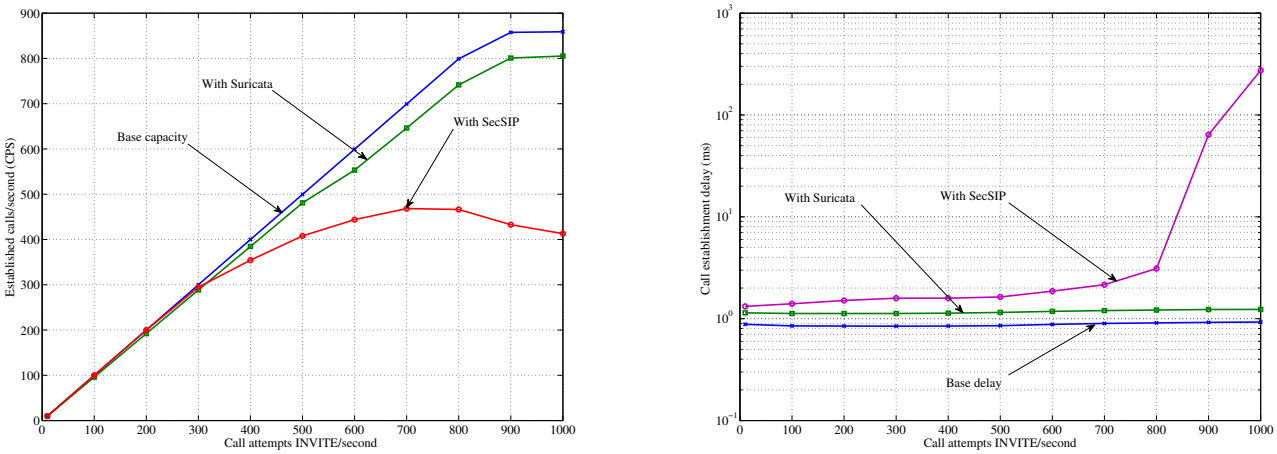
Figure 5(a) shows the number of established calls between two user agents through an OpenSIPS SIP proxy with and without a deployed prevention runtime running VeTo rules. We observe that the base capacity for the SIP proxy without SecSIP is around 800 CPS. The call handling capacity is dropped to 500 CPS when enabling the Suricata tool and it is dropped to 300 CPS when enabling SecSIP. SecSIP introduces more overhead than Suricata because it inspects and parses more deeply SIP messages when applying prevention rules. However, Suricata only applies string based regular expression matching over the intercepted message.

Figure 5(b) depicts the call establishment delay which is measured on the caller (UAC) between the sending of the INVITE message and the receipt of its respective ACK. We observe a mean delay close to 0.8 ms while the SIP traffic goes only through the SIP proxy. The delay increases a little more when deploying Suricata and it remains close to 1 ms. When we deploy SecSIP, delays remain between 1 and 10 ms with call attempts lower than 800 INVITE/second. However, the delay becomes more important under heavy call attempts. It is close to 100 ms under 900 concurrent call attempts. The overhead introduced by SecSIP appears important than Suricata but remains acceptable since the establishment delay remains far below the transaction expiration delay of 32 seconds. We have also verified that each INVITE and 200 OK messages and their respective responses 100 Trying and ACK messages fit within a time window lower than 500 ms.

2) Effect of the number of VeTo blocks: In a second experiment, we varied the number of VeTo prevention blocks to be handled by SecSIP between 10 and 1000 prevention blocks with a step of 100. In each step, we duplicate the IllegalSize prevention block depicted in Rule 7. We fixed the number of call attempts to 300 INVITE/second.

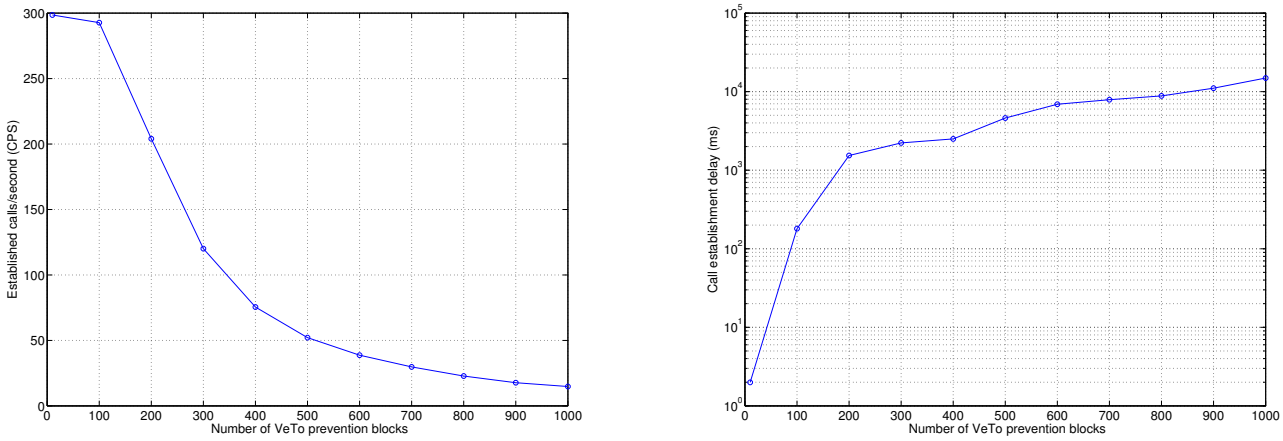
We observe in Figure 6 that the number of established calls decreases drastically and the average call establishment delay increases exponentially when the number of deployed VeTo blocks increases. These results represent the worst cases since for each SIP message all the deployed VeTo blocks are applied on it since we have not activated the context feature provided by the VeTo language. However in a real scenario where context is active, the performance should be better since only few prevention blocks will be activated on an intercepted SIP message according to its target.

3) Flooding prevention: In this scenario, we deployed on SecSIP the prevention block depicted in Rule 9. This block prevents a source from flooding a target with a high INVITE message rate. The flow of INVITE messages originates from a single source with an increasing rate of 1 every 2 seconds. The starting rate is fixed at 1 INVITE/second. Figure 7 contains the measurements. We observe that the SecSIP runtime only allows a number of concurrent messages equal to 10. When an attack source exceeds this number and its traffic rate becomes high, then it is blocked and every incoming message from the attack source is dropped. The attack source has to wait a period of time before any of its incoming traffic will be forwarded again.



(a) (b)

Fig. 5. Number of established calls (a) and call establishment delays (b) under increasing number of calls in terms of INVITE/second, without and with a deployed SecSIP and Suricata runtimes.



(a) (b)

Fig. 6. Number of established calls (a) and call establishment delays (b) under increasing number of VeTo prevention blocks with a deployed SecSIP runtime.

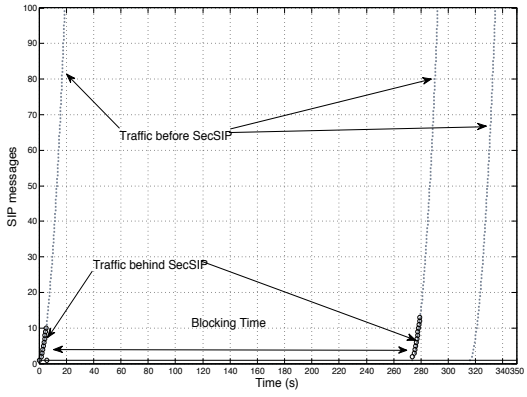


Fig. 7. Prevention against an INVITE messages based flooding from a single source.

VIII. Conclusions

SecSIP is a framework that prevents from the exploitation of existing vulnerabilities in SIP-based networks. SecSIP uses a domain specific language called VeTo to code prevention specifications. The VeTo language is stateful and it couples rule and an event-based approach. It includes instructions to record and maintain SIP protocol histories over messages, transactions and dialogs. Our approach was extended to automatically generate VeTo specifications using a genetic algorithm. The produced specifications are generated from network traces of messages known as exploiting a vulnerability. Such messages may be identified by a fuzzing tool or provided by security advisors and/or administrators. This paper described the overall framework to specify, generate and execute VeTo prevention schemes. We implemented and evaluated the different components of the framework. We showed that the SecSIP overhead, both in terms of a call establishment delay and the number of established calls, remains sufficiently low to operate efficiently in most enterprise networks. We demonstrated the usage of the VeTo language through several types of vulnerabilities behind flooding and DoS attacks.

Current VeTo specifications only address known vulnerabilities since our aim was to protect a SIP network from discovered and unpatched vulnerabilities. Unknown vulnerabilities can be expressed in VeTo and deployed as soon as they become known. Other techniques, like software testing and binary analysis have to be used in conjunction to prevent from unknown and zero-day vulnerabilities. In future work, we hope to demonstrate formal modeling techniques in order to verify conflicts between VeTo rules and check their consistency, completeness and compactness. In particular, we are interested in tree automata techniques to automatically analyze and validate new VeTo rules before integrate them with previous existing rules.

REFERENCES

[1] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "SIP: Session Initiation Protocol," RFC 3261 (Proposed Standard), Jun. 2002.

[2] G. Ormazabal, S. Nagpal, E. Yardeni, and H. Schulzrinne, "Secure sip: A scalable prevention mechanism for dos attacks on sip based voip systems," in *Principles, Systems and Applications of IP Telecommunications. IPTComm 2008, Heidelberg, Germany, July 1-2, 2008. Revised Selected Papers*, 2008, pp. 107–132.

[3] A. Keromytis, *Voice over IP Security: A Comprehensive Survey of Vulnerabilities and Academic Research*. Springer, 2011, springerBriefs in Computer Science.

[4] H. Abdelnur, O. Festor, and R. State, "KiF: A stateful sip fuzzer," in *1st International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, ACM, Ed., July 2007.

[5] E. Rescorla, "Security holes... who cares?" in *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2003, pp. 75–90.

[6] B. Caswell, J. Beale, J. C. Foster, and J. Faircloth, *Snort 2.0 Intrusion Detection*. Syngress, May 2003.

[7] J. Fielder, T. Kupta, S. Ehlert, T. Magedanz, and D. Sisalem, "VoIP Defender: Highly scalable sip-based security architecture," in *International Conference on Principles, Systems and Applications of IP Telecommunications (IPTComm)*, ACM, Ed., New York, USA, 19-20 July 2007, pp. 11–17, iSBN: 978-1-60558-006-7.

[8] Y.-S. Wu, S. Bagchi, S. Garg, N. Singh, and T. Tsai, "Scidive: A stateful and cross protocol intrusion detection architecture for voice-over-ip environments," in *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 433–442.

[9] H. Sengar, D. Wijesekera, H. Wang, and S. Jajodia, "Voip intrusion detection through interacting protocol state machines," *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pp. 393–402, 2006.

[10] E. Chen, "Detecting dos attacks on sip systems," *1st IEEE Workshop on VoIP Management and Security*, pp. 53–58, April 2006.

[11] A. Lahmadi and O. Festor, "Veto: An exploit prevention language from known vulnerabilities in sip services," in *IEEE/IFIP Network Operations and Management Symposium, NOMS, 19-23 April 2010, Osaka, Japan*. IEEE, 2010, pp. 216–223.

[12] ——, "Secsip: A stateful firewall for sip-based networks," in *11th IFIP/IEEE International Symposium on Integrated Network Management, IM'09, Long Island, New York, USA*, June 2009.

[13] S. T. Eckmann, G. Vigna, and R. A. Kemmerer, "STATL: an attack language for state-based intrusion detection," *Journal of Computer Security*, vol. 10, no. 1-2, pp. 71–103, 2002.

[14] H. J. Wang, C. Guo, D. R. Simon, and A. Zugenmaier, "Shield: vulnerability-driven network filters for preventing known vulnerability exploits," *SIGCOMM Comput. Commun. Rev.*, vol. 34, no. 4, pp. 193–204, 2004.

[15] M. V. Hayden, "National information systems security glossary," September 2000. [Online]. Available: http://security.isu.edu/pdf/4009.pdf. Last visit december 2010.

[16] D. Schwartz and J. Barkan, "End-to-end route management in the session initiation protocol," http://tools.ietf.org/html/draft-schwartz-sip-routing-managment-00, February 2006.

[17] R. Zhang, X. Wang, X. Yang, and X. Jiang, "Billing attacks on sip-based voip systems," in *WOOT '07: Proceedings of the first USENIX workshop on Offensive Technologies*. Berkeley, CA, USA: USENIX Association, 2007, pp. 1–8.

[18] H. Abdelnur, R. State, I. Chrisment, and C. Popi, "Assessing the security of VoIP Services," in *Integrated Network Management, IM 2007. 10th IFIP/IEEE International Symposium on Integrated Network Management, Munich, Germany*. IEEE, May 2007, pp. 373–382.

[19] D. Geneiatakis, T. Dagiuklas, G. Kambourakis, C. Lambrinoudakis, S. Gritzalis, K. Ehlert, and D. Sisalem, "Survey of security vulnerabilities in session initiation protocol," *Communications Surveys & Tutorials, IEEE*, vol. 8, no. 3, pp. 68–81, 2006.

[20] A. Habib, M. M. Hefeeda, and B. K. Bhargava, "Detecting service violations and dos attacks," in *In Proceedings of 2003 Internet Society Symposium on Network and Distributed System Security (NDSS03)*, 2003, pp. 177–189.

[21] J. M. Kizza, *Guide to Computer Network Security*. Springer Publishing Company, Incorporated, 2008.

[22] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Comput. Netw.*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[23] S. Ehlert, D. Geneiatakis, and T. Magedanz, "Survey of network security systems to counter sip-based denial-of-service attacks," *Computers & Security*, vol. 29, no. 2, pp. 225–243, 2010.

[24] G. Vigna, S. Eckmann, and R. Kemmerer, "Attack languages," in *Proceedings of the IEEE Information Survivability Workshop*, 2000.

[25] N. Borisov, D. J. Brumley, and H. J. Wang, "A generic application-level protocol analyzer and its language," in *14th Annual Network & Distributed System Security Symposium*, 2007.

[26] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State transition analysis: A rule-based intrusion detection approach," *IEEE Transactions on Software Engineering*, vol. 21, pp. 181–199, 1995.

[27] R. Smith, C. Estan, S. Jha, and S. Kong, "Deflating the big bang: fast and scalable deep packet inspection with extended finite automata," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 4, pp. 207–218, 2008.

[28] D. Geneiatakis, G. Kambourakis, C. Lambrinoudakis, T. Dagiuklas, and S. Gritzalis, "A framework for protecting a sip-based infrastructure against malformed message attacks," *Comput. Netw.*, vol. 51, no. 10, pp. 2580–2593, 2007.

[29] VoIPSA.org, "VOIPSEC mailing list on VoIP security issues," http://voipsa.org/mailman/listinfo/voipsec_voipsa.org, January 2009.

[30] K. Man, K. Tang, and S. Kwong, "Genetic algorithms: concepts and applications [in engineering design]," *Industrial Electronics, IEEE Transactions on*, vol. 43, no. 5, pp. 519 –534, Oct. 1996.

[31] R. Adaikkalavan and S. Chakravarthy, "Snoopib: interval-based event specification and detection for active databases," *Data Knowl. Eng.*, vol. 59, pp. 139–165, October 2006.

[32] R. Gayraud and O. Jacques, *SIPp Reference Manual*, June 2008, http://sipp.sourceforge.net/.

[33] Open Information Security Foundation, "Suricata IDS/IPS," http://www.openinfosecfoundation.org/, September 2011.

Abdelkader Lahmadi is an associate professor in the ENSEM engineering school at Nancy University, France and a member of the MADYNES research team. He holds an engineering degree from ENSI in Tunis and obtained a Ph.D degree in computer science in 2007 from Henri Poincare University - Nancy. His research interests include security monitoring, embedded systems and protocols fuzzing.

Olivier Festor is a research director at Inria. He has a Ph.D. degree (1994) and an Habilitation degree (2001) from Henri-Poincare University, France. His research interests are in the design of algorithms and architectures for automated monitoring and security management of large-scale networks. He is co-chair of the IRTF NMRG and IFIP WG6.6 and serves in the committees as well as in the editorial boards of the major international conferences and journals in network and service management.